Introduction to Programming

Block Tutorial C/C++

Michael Bader

Master's Program
"Computational Science and Engineering"

C/C++ Tutorial – Overview

- From Maple to C
- Variables, Operators, Statements
- Functions: declaration, definition, parameters
- Arrays and Pointers
- From Pointers to Dynamical Data Structures
- Object Oriented Programming: From C to C++

Tuesday Lesson – Overview

- C functions: definition and function calls
- passing parameters
- declaration vs. definition: header files
- local variables
- recursion

Functions in C

Functions are the basic units of every C program:

- a C program is a sequence of functions
- in C, functions cannot be nested (i.e., there are no "local" functions)
- functions execute certain subtasks of a program (compute a value, print something, change variables, etc.)
- every program starts with a call to `main()`
- the sequence of function calls is predetermined (but may depend on user input)
- not all functions are necessarily used during the execution of a program.

Defining functions in C

Actually, we already know how to define a basic function in C:

```
main() {
   printf("Hello World\n");
}
```

`main` is a function we've been using quite extensively.

General Syntax of a function definition:

```
[type] <name>([list of parameters]) {
   [declaration of variables]
   <statements>
}
```

Defining functions in C – return type

- Like a Maple procedure, a C function can return a value.
- type of return value is specified before the function name

Example: `main` is a function that should return an integer value:

```
int main() {
   printf("Hello World\n");
   return 0;
}
```

The return value of `main()` is passed to the operating system.
A return value of 0 indicates "everything OK".

Defining functions in C – return statement

Syntax: `return <expression>;`

- the expression stated after `return` defines the return value of the function
- `return` terminates the function call (like in Maple)

Example:

```
int main() {
   float x;
   scanf("%f",&x) ;
   if (x<0) return 1; /* indicate error! */
   printf("%f\n", sqrt(x) );
   return 0;
}
```

Functions without return value – `void`

Sometimes, a function does not need to return any value, e.g. we almost never use the return value of `main`.

To indicate the absence of a return value explicitly, we can use the special type `void`:

```
void main() {
   float x;
   scanf("%f",&x) ;
   if (x>=0)
      printf("%f\n", sqrt(x) );
   else
      printf("Error: input should be >= 0.\n");
}
```

As we can see, a function without return value is not necessarily useless!

Functions with parameters

Syntax:

```
<functiondef> = [type] <name>([list of parameters]) { ...
<list of parameters> = <type> <variable>, <type> <variable>, ...
```

Example:

```
int main(int argc, char** argv) {
    ...
}
```

- actually, `main` has parameters, too!
- the parameters are passed "from the command line"
- `argc` stands for "argument counter" (and integer number)
- `argv` stands for "argument vector" (we'll learn about the `*`-types later)

Function calls

Function calls in C work just like we know them from Maple:

```
float power(float base, int exponent) {
   int i;
   float result = 1.0;
   for(i=1;i<=exponent;i++) result *= base;
   return result;
}
int main() {
   int n;
   float x, result;
   scanf("%f %d",&x,&n);
   result = power(x,n);
   printf("%g\n",result);
}
```

Function calls – "call by value"

At each function call, the values of the expressions in the function call have to be transferred to the function parameters.

The technique adopted in C is called **"call by value"**:

- first the expressions in the function call are evaluated
- the respective values are then assigned to the parameters
- after termination, the parameters are no longer accessible
- changing the values of parameters inside a function does not affect the expressions in the function call (even if they are simple variables)

Function calls – "call by value"

Example:

```
float power(float x, int n) {
   float res = 1.0;
   while (n>=1} { res *= x; n--; }
   return res;
}

int main() {
   float pi=3.1415;
   int two=2;
   printf("%f^%d = %g\n", pi, two, power(pi,two) );
}
```

On the Correct Order of Function Definitions

Example: Which of the two functions should be defined first?

```
int odd(int n) {
   if (n==1) return 1;
   if (n==0) return 0;
   return even(n-1);
}
```

```
int even(int n) {
   if (n==0) return 1;
   if (n==1) return 0;
   return odd(n-1);
}
```

- `odd` needs to know about `even` first
- `even` needs to know about `odd` first

$\Rightarrow$ whatever function is defined first, the compiler will cast an error!
(at least, it should do so . . .)

On the Correct Order of Function Definitions (2)

Problem: Which of the two functions should be defined first?

```
int odd(int n) {
   if (n==1) return 1;
   if (n==0) return 0;
   return even(n-1);
}
```

```
int even(int n) {
   if (n=0) return 1;
   if (n=1) return 0;
   return odd(n-1);
}
```

- Actually, `odd` does not need to know exactly what `even` does.
- it **does** need to know how to call it, however!

$\Rightarrow$ it is sufficient to **declare the signature** of the function

Function Declarations

Syntax:

```
[type] <name>([parameters]);
```

- declare signature of a function, i.e. how to call it
- define later, what the function actually does (”function definition”)

$\Rightarrow$ no longer necessary to define functions in any specific order

Example:

```
int odd(int n);
int even(int n);
int odd(int n) {
    if (n==1) return 1;
    if (n==0) return 0;
    return even(n-1);
}
int even(int n) {
    if (n=0) return 1;
    if (n=1) return 0;
    return odd(n-1);
}
```

Header Files

Function declarations are often put into separate *header files*:

Header File: `evenodd.h`

```
int odd(int n);
int even(int n);
```

Program File: `evenodd.c`

```
#include "evenodd.h"
int odd(int n) {
    ...
}
int even(int n) {
    ...
}
```

Double quotes in `#include "evenodd.h"` hint that `evenodd.h` is a **user defined** header file!

Large Projects

The concept of header files leads to a lot of advantages:

- projects may be partitioned into several source files; each source file has a separate header file
- projects can be divided into several modules
- modules can be compiled separately
- modules can be re-used for other projects

$\Rightarrow$ Linking object code to executable becomes more important (check for duplicate functions, etc.)

Local Variables

Definition: Every variable declared inside a function definition or a block statement is called a *local variable*.

Properties:

- a local variable is only defined within the block it was defined
- it is also defined in any block statement inside of this block
- defining a local variable "shadows" a variable of the same name that was defined in an outer block
- a local variable is destroyed as soon as the corresponding block is finished
- in recursive programming, each function call has its own set of local variables

Local Variables – Example

Printing Pascal's Triangle:

```
void triangle_line(int n) {
   int i;
   for(i=0;i<=n;i++)
      printf("%d ",binom(n,i));
}
void triangle(int n) {
   int i;
   for(i=0;i<=n;i++) {
      triangle_line(i);
      printf("\n");
   }
}
```

Local Variables and Parameters

Every parameter of a function can be considered as a local variable:

- a parameter is only defined within its function
- the value of a parameter may be changed inside the function, but is lost as soon as the function call is finished
- defining a local variable "shadows" a parameter of the same name (there are hardly any situations where this is useful)
- in recursive programming, every function call has its own set of parameters

Local Variables – Recursive Example

Pascal's Triangle, recursive implementation:

```
void triangle(int n) {
   int i;
   if (n>0) triangle(n-1);
   for(i=0;i<=n;i++)
      printf("%d ",binom(n,i));
   printf("\n");
}
```

Exercise:

Draw a diagram that outlines the function calls together with the respective values of `n` and `i`.

Global Variables

Definition: A variable declared outside of all function definitions is called a *global variable*.

Properties:

- a global variable can be accessed and modified in any function unless it is shadowed by a local variable or a parameter
- a global variable exists as long as the program is running
- defining a local variable "shadows" a variable of the same name that was defined in an outer block
- global variables should be avoided as far as possible!

Global Variables – Example

Implementation of a stack:

```
int stack[1000];
int stackpointer=0;

void push(int value) {
   if (stackpointer==1000)
      exit(1) ; /* stack overflow! */
   else
      stack[stackpointer++] = value;
}
int pop() {
   return stack[--stackpointer];
}
```

Local Variables, Global Variables and Parameters – Implementation

There are usually three separate areas of memory used for a program:

- the **program stack** holds all local variables and parameters, only the variables and parameters of the latest function call are accessible;
- the **program heap** holds all variables that are created dynamically, e.g. using `malloc();`
- global variables and constants can be stored together with the program code or be placed in the program heap.